

# Comparison and evaluation of back-translation algorithms for static single assignment forms<sup>☆</sup>

Masataka Sassa<sup>\*</sup>, Yo Ito<sup>1</sup>, Masaki Kohama<sup>2</sup>

*Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1, O-okayama, Meguro-ku, Tokyo 152-8552, Japan*

Received 13 May 2006; received in revised form 3 January 2007; accepted 6 March 2007

---

## Abstract

The static single assignment form (SSA form) is a popular intermediate representation in compilers. In the SSA form, the definition of each variable textually appears only once in the program by using a hypothetical function called a  $\phi$ -function. Because these functions are nonexecutable, it is necessary to delete the  $\phi$ -functions and return the SSA form to the normal form before code generation. This conversion is called *SSA back-translation*.

Two major algorithms exist for SSA back-translation. One by Briggs et al., the other by Sreedhar et al. To date, there has been almost no research that compares these SSA back-translation algorithms.

In this paper, we clarify the merits and demerits of these algorithms. We also propose an improvement to Briggs' algorithm. We then compare the three methods through experiments using the SPEC benchmarks.

Our experiments show that in most cases, Sreedhar's method is the more favorable. The efficiency of its object code is better than that from Briggs' method by a few percent in general, up to a maximum of 28%. The experiments have also clarified several characteristic features of these methods.

© 2007 Elsevier Ltd. All rights reserved.

*Keywords:* Static single assignment form; Back-translation; Intermediate language; Machine-independent program transformation; Compiler

---

## 1. Introduction

Optimizations in compilers are crucial for the efficient execution of large programs. The static single assignment form (SSA form) [1] is a popular intermediate representation of code in compilers because of its simplicity for dataflow analysis and the efficiency of optimization algorithms [2–6].

In the SSA form, the definition of each variable textually appears only once in the program by using a hypothetical function called a  $\phi$  ( $\phi$ )-function. Because the  $\phi$ -functions appearing in the SSA form are nonexecutable, it is necessary to delete the  $\phi$ -functions and return the SSA form to the normal form before code generation. This translation is called *SSA back-translation*.

---

<sup>☆</sup> Earlier versions of parts of this article appeared at IPSI-2004 Prague Meeting and at Transactions of Information Processing Society of Japan—Programming in Japanese.

<sup>\*</sup> Corresponding author. Tel.: +81 3 5734 3228; fax: +81 3 5734 3210.

*E-mail address:* [sassa@is.titech.ac.jp](mailto:sassa@is.titech.ac.jp) (M. Sassa).

<sup>1</sup> Currently with NEC Corporation.

<sup>2</sup> Currently with Fuji Photo Film Co., Ltd.

The naive SSA back-translation algorithm by Cytron et al. [1] has several critical problems, and it behaves incorrectly when it is applied to SSA forms that have been transformed by some optimizations [7]. To remedy this, Briggs et al. proposed a different back-translation algorithm [7,8]. Later, Sreedhar et al. proposed another back-translation algorithm based on a completely different approach [9]. (Hereafter, those methods are simply called Briggs' method and Sreedhar's method.) The former is adopted in many compilers that use the SSA form, such as Marmot [2], gcc [3], Machine SUIF [5] and Scale [6]. However, the latter is not widely adopted, except in [10], probably because it was proposed a few years after Briggs' method.

Briggs' method replaces  $\phi$ -functions by copy statements. This increases the number of copy statements, but the authors claim that those copy statements can be deleted by performing a coalescing pass. Sreedhar's method accomplishes a kind of coalescing to the target (left-hand side) and parameters of  $\phi$ -functions, which we call "uniting".

In Sreedhar's method, uniting may augment the length of live ranges of variables associated with  $\phi$ -functions. This may increase the register pressure and thus may cause spilling of registers if the number of allocatable registers is relatively small. This may harm execution efficiency.

On the other hand, in Briggs' method, many copy statements are inserted, which may increase the number of executed instructions. However, it was thought that the undesirable situation of Sreedhar's method would not occur if coalescing was performed properly in a later phase.

Thus, these two SSA back-translation algorithms have different characteristics, and the choice of algorithm may affect the runtime efficiency of the object code after register allocation and code generation. However, there have been no careful comparisons of these algorithms to date. Sreedhar et al. gave some theoretical discussion of whether the result of their translation has minimum code length, but they could not show the minimality. Of course, minimality of code itself does not assure minimality of the runtime efficiency of the object code. Moreover, Sreedhar et al. offered no comparisons with other SSA back-translation algorithms [9]. To sum up, to date there has been no empirical comparison of back-translation algorithms, especially considering register allocation.

In this paper, we clarify the merits and demerits of these two algorithms for SSA back-translation. We also propose an improvement of Briggs' algorithm. We implemented Briggs' algorithm, its improvement and Sreedhar's algorithm on the same compiler, and experimented using the SPEC benchmarks by changing the number of allocatable registers. We evaluated the influence of coalescing, the relationship with the number of allocatable registers and the relationship with the set of optimizations, which gave insights into the results.

The results show that in Briggs' method, many copy statements that cannot be coalesced remain, which falls short of the original expectations. These copy statements not only cause extra execution time, but may also consume registers wastefully. The combination of optimizations does not much affect the differences between the back-translation algorithms. The main result of the experiment is that, in most cases, Sreedhar's method is the more favorable. The efficiency of its object code is better than that of Briggs' method by a few percent in general and by up to 28% at maximum.

## 2. Static single assignment form

The SSA form [1,11] is an internal representation of programs in which indices are attached to variables so that the definition of each variable in a program becomes textually unique. At a joining point of the control flow graph (CFG) reached by two or more different definitions of a variable, a hypothetical function called a  $\phi$ -function is inserted so that these multiple definitions are merged into one (Fig. 1).

The statement " $x3 = \phi(x1:L1, x2:L2);$ " in Fig. 1(b) means that  $x3$  is assigned the value of  $x1$  when the control comes from L1, and the value of  $x2$  when the control comes from L2.

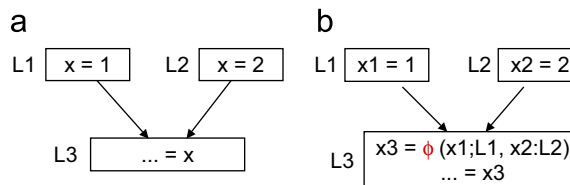


Fig. 1. The static single assignment form. (a) Normal form and (b) SSA form.

Dataflow analysis and optimization for sequential execution can be compacted using the SSA form, because it clarifies the relations between definitions and uses of variables.

### 2.1. SSA translation

Let us call the conventional representation form before translating into SSA form the *normal form*. SSA translation translates the normal form into the SSA form. The SSA translation algorithm proposed by Cytron et al. [1] and that by Sreedhar et al. [12] are well known.

## 3. SSA back-translation

The SSA form includes the hypothetical  $\phi$ -functions and cannot be directly translated into assembly code or machine code. Therefore, translation back to normal form, which deletes the  $\phi$ -functions, is necessary before code generation.

We call the translation from SSA form into normal form *SSA back-translation*.

### 3.1. Naive algorithm for SSA back-translation

Cytron et al. [1] presented a naive algorithm for SSA back-translation. The role of  $\phi$ -function, which merges several definitions of variables, ceases after it is replaced by copy statements in the predecessor blocks. Therefore, the back-translation inserts copy statements for variables used in the  $\phi$ -function into the predecessor blocks of the basic block where the  $\phi$ -function resides, and then deletes the  $\phi$ -function. This gives the normal form. Fig. 2 shows an example of the naive SSA back-translation.

In Fig. 2(a), variables  $x_1$  and  $x_2$  used in the parameters of the  $\phi$ -function in block L3 are defined in blocks L1 and L2, respectively. For example, if control goes from L1 to L3, the value of  $x_3$  becomes the value of  $x_1$ . The naive method for SSA back-translation puts the definitions of variable  $x_3$ , originally defined in the  $\phi$ -function of block L3, at the end of the instructions of L1 and L2, which are the predecessor blocks of L3. For example, it puts the copy statement “ $x_3 = x_1$ ” at the end of block L1. Then it deletes the  $\phi$ -function. This produces the result shown in Fig. 2(b). This algorithm is simple, but it introduces problems as shown below. We therefore call it the *naive back-translation algorithm*.

### 3.2. Problems of the naive back-translation algorithm

It has been pointed out that the naive back-translation algorithm does not work correctly in some cases [7,9]. There are two main factors that make the naive back-translation algorithm cause such problems.

The first problem is related to the destruction of live ranges when inserting copy statements. An example of such an incorrect behavior is the so-called *lost copy problem* and is shown in Fig. 3.

Fig. 3(a) shows the usual SSA form. After applying copy propagation optimization to this SSA form, “ $y = x_1$ ” in block 2 is deleted, and  $y$  in block 3 is replaced by  $x_1$ , as in Fig. 3(b). This SSA form is correct. Then, if we apply the above naive algorithm to this optimized SSA form, it inserts copy statements at the end of each predecessor block. That is, it inserts a copy statement “ $x_1 = x_0$ ” at the end of block 1, and “ $x_1 = x_2$ ” at the end of block 2 and deletes the  $\phi$ -function (Fig. 3(c)). However, this is incorrect. The value returned by “*return x\_1*” is now always 2, which is different from the original SSA form program. The reason for this error is that the method inserts the copy statement “ $x_1 = x_2$ ” at a point where  $x_1$  is live, destroying the current value of  $x_1$ .

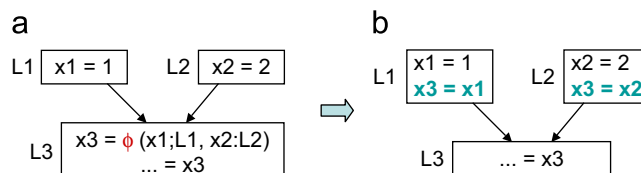


Fig. 2. Naive SSA back-translation. (a) SSA form and (b) normal form.

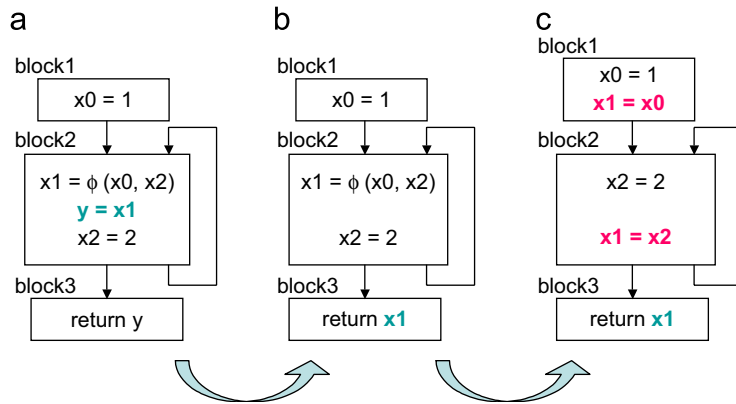


Fig. 3. The lost copy problem. (a) copy propagation, (b) back-translation by native method and (c) not correct.

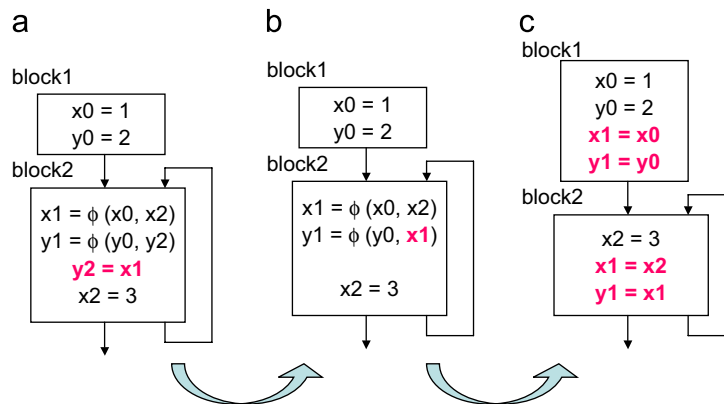


Fig. 4. The simple ordering problem. (a) copy propagation, (b) back-translation by native method and (c) not correct.

Note that the lost copy problem alone can be handled by the naive algorithm if critical edges are removed. However, because there are also other problems in the naive algorithm, the revised algorithms to be presented in Section 4 handle this without edge splitting.

A second problem with the naive SSA back-translation algorithm occurs when there are multiple  $\phi$ -functions in the same block. Examples can be seen in the so-called *simple ordering problem* and the *swap problem*, which we do not discuss further [7,9].

The simple ordering problem is shown in Fig. 4. Fig. 4(a) is the usual SSA form. After applying copy propagation optimization to this SSA form, “ $y2 = x1$ ” of block 2 is deleted, and “ $y1 = \phi(y0, y2)$ ” is replaced by “ $y1 = \phi(y0, x1)$ ” (Fig. 4(b)). This SSA form is correct. Then, if we apply the naive algorithm to this optimized SSA form, it inserts the copy statements “ $x1 = x0$ ” and “ $y1 = y0$ ” at the end of block 1, and “ $x1 = x2$ ” and “ $y1 = x1$ ” at the end of block 2, and we obtain the SSA form in Fig. 4(c). However, this is incorrect. The value of “ $y1$ ” at the exit of block 2 is now always 3, which is different from the original SSA form program.

The semantics of the SSA form require that all  $\phi$ -functions in the same block must be regarded as *simultaneous assignments*. For example, in the SSA form in Fig. 4(b), assignments to  $x1$  and  $y1$  must be considered as occurring simultaneously. The reason for the error in the back-translation here is that the naive method inserts copy statements without considering the simultaneous assignment property of the  $\phi$ -function. As a result, a dependency for  $x1$  is introduced among the copy statements inserted in block 2. By executing these statements sequentially, program behavior differs from the original.

### 4. Two major algorithms for SSA back-translation

Briggs’ and Sreedhar’s algorithms were proposed to remedy the problems presented in Section 3.2, such as the lost copy problem, the simple ordering problem and the swap problem.

There are also other algorithms, such as those of Morgan [13] and Rastello [14]. The former can be thought of as a subset of Briggs’ algorithm (cf. Section 8.1) and the latter is discussed below (cf. Section 8.3).

#### 4.1. The algorithm of Briggs et al.

The SSA back-translation algorithm by Briggs et al. [7,8] extends the naive back-translation algorithm to perform a safe translation (Fig. 5).

Of the problems with the naive back-translation algorithm, we concentrate here on the “lost copy problem” described in Section 3.2.

The problem with the naive translation in Fig. 3 arose because the value of  $x_1$  in block 2 is destroyed by the insertion of “ $x_1 = x_2$ ”. Briggs’ method inserts an assignment to a temporary variable, “ $temp = x_1$ ”, at the entry of block 2 to save the value of  $x_1$  at this point into  $temp$ , and replaces the use of  $x_1$  in later blocks by  $temp$ , as in Fig. 5(b). In this way, Briggs’ method realizes a correct SSA back-translation by inserting copy statements such as “ $temp = x_1$ ” to remedy the problems of the naive method.

We see from this example that this SSA back-translation algorithm may insert many copy statements, both the copy statements inserted by the naive method and those inserted to avoid these critical problems.

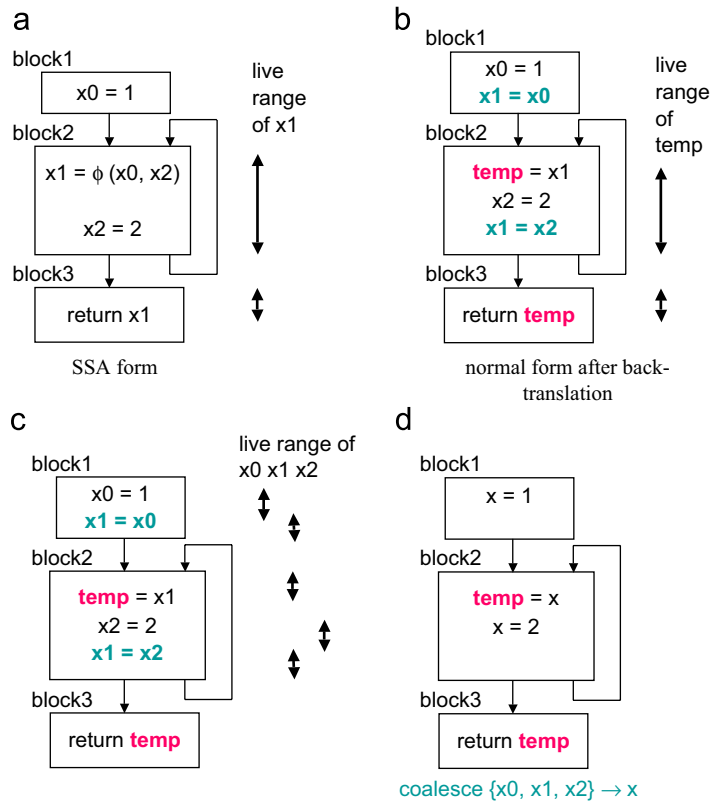


Fig. 5. SSA back-translation by Briggs et al. (lost copy problem). (a) SSA form, (b) normal form after back-translation, (c) normal form after back-translation (same as (b)) and (d) after coalescing.

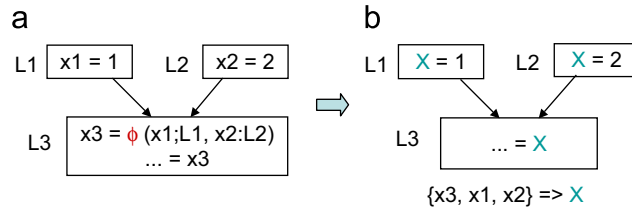


Fig. 6. SSA back-translation by Sreedhar et al.—principle. (a) SSA form and (b) normal form.

However, Briggs et al. claimed that *coalescing* the live ranges (coalescing [15] is usually performed in the register allocation phase) after the back-translation can eliminate most of these copy statements. This is illustrated in Fig. 5(c) and (d). Because the live ranges of  $x0$ ,  $x1$  and  $x2$  that are connected by copy statements do not interfere, as seen in Fig. 5(c), they can be coalesced into a single variable  $x$  as shown in Fig. 5(d).

We will also perform coalescing in the experiments in Sections 6 and 7.

#### 4.2. Algorithm of Sreedhar et al.

The SSA back-translation algorithm by Sreedhar et al. [9] uses a completely different approach from the naive algorithm or Briggs' algorithm.

Rather than insert copy statements to delete  $\phi$ -functions, Sreedhar's algorithm checks whether there is interference between the live ranges of the parameters of each  $\phi$ -function. Here we regard the left-hand side variable of the  $\phi$ -function as another parameter. If there is such interference, the algorithm renames some of the parameters by inserting copy statement(s) so that finally there is no more interference of live ranges between parameters. Then, it replaces all the parameters of the  $\phi$ -function by a single variable and deletes the  $\phi$ -function.

A very simple example of Sreedhar's algorithm is shown in Fig. 6. In Fig. 6(a), there is no interference between the parameters  $x3$ ,  $x1$  and  $x2$  (including the left-hand side variable  $x3$ ) of the  $\phi$ -function, so all the parameters  $x3$ ,  $x1$  and  $x2$  can be replaced by the same variable  $X$  and the  $\phi$ -function can be deleted, as shown in Fig. 6(b). In this example, no copy statements are inserted. (We can regard this replacement to the same variable as a kind of coalescing. In this paper, we call this *uniting*.) There is no need to insert copy statements in this case, as opposed to the naive or Briggs' algorithm.

A more complex example can be found in the treatment of the lost copy problem. Fig. 7(a) is the same SSA form as in Fig. 3(b). Consider the  $\phi$ -function in block 2. It has three parameters,  $x1$ ,  $x0$  and  $x2$ . Because there is interference of live ranges between  $x1$  and  $x2$ , we replace  $x1$  by  $x1'$  and insert a copy " $x1 = x1'$ ", producing Fig. 7(b). Now there is no interference between the live ranges of parameters  $x1'$ ,  $x0$  and  $x2$  of the  $\phi$ -function. We therefore replace all the parameters of the  $\phi$ -function by a single variable and delete the  $\phi$ -function. In this example, we replace  $x1'$ ,  $x0$  and  $x2$  by  $X$ , and obtain the normal form shown in Fig. 7(c).

The main part of Sreedhar's algorithm resides in the treatment of the interference between the live ranges of the parameters of a  $\phi$ -function. In general, if there is such interference, uniting these parameters is not possible. In that case, we must remove the interference caused by these parameters. Sreedhar's algorithm accomplishes this by rewriting the  $\phi$ -function (Fig. 8). This is done by combining the following processes. They have the effect of shortening the live ranges of the parameters of the  $\phi$ -function, and they finally remove the interference.

- The left-hand side variable of a  $\phi$ -function is called a *target*. The target of a  $\phi$ -function is considered live at the entry of the block where the  $\phi$ -function is placed. Therefore, to minimize the live range of the target, perform rewriting as in Fig. 8(b).
- A parameter on the right-hand side of a  $\phi$ -function is called a *source*. The source of a  $\phi$ -function is considered live at the exit of the block that is the predecessor to the block where the  $\phi$ -function is placed. (It is not live at the entry of the block where the  $\phi$ -function is placed.) Therefore, to minimize the live range of the source, perform rewriting as in Fig. 8(c).

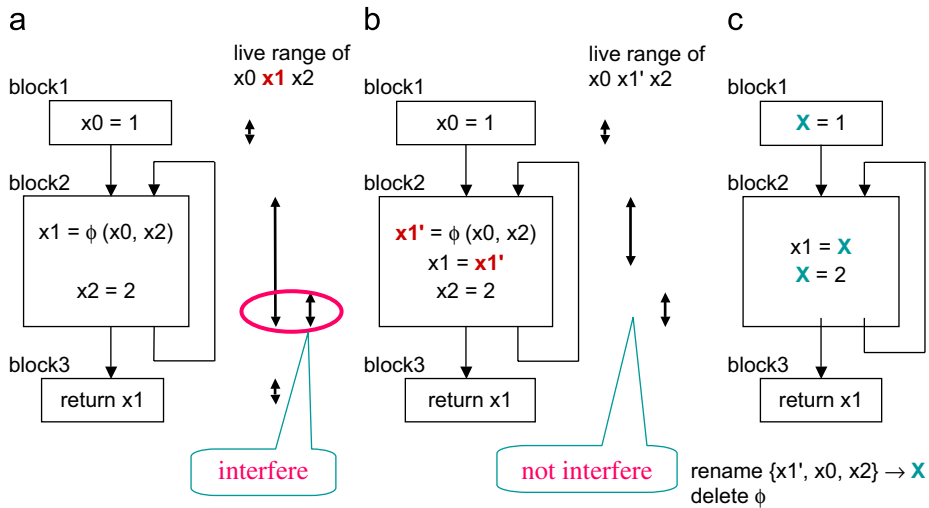


Fig. 7. SSA back-translation by Sreedhar et al. (lost copy problem). (a) SSA form, (b) eliminating interference and (c) normal form after back-translation.

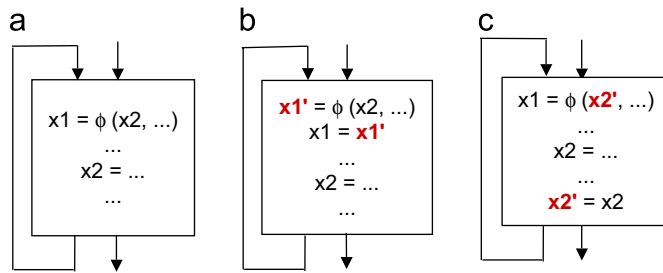


Fig. 8. Sreedhar's rewriting of the  $\phi$ -function. (a) SSA form, (b) rewrite target and (c) rewrite source.

Note again that in Sreedhar's algorithm, no insertion of copy statements for *all* parameters of the  $\phi$ -functions is generally required, in contrast to Briggs' algorithm. Therefore, fewer copy statements are generally inserted.

In Sreedhar's algorithm, any copy statement that it inserts cannot be eliminated by the standard interference-graph-based coalescing algorithm [9]. Note also that Sreedhar's algorithm can perform coalescing based on the SSA form at back-translation time [9].

## 5. Problems and proposal for improvement

The two algorithms for SSA back-translation presented so far have some weaknesses. We discuss them and propose an improvement in this section.

### 5.1. A preliminary comparison of the two algorithms

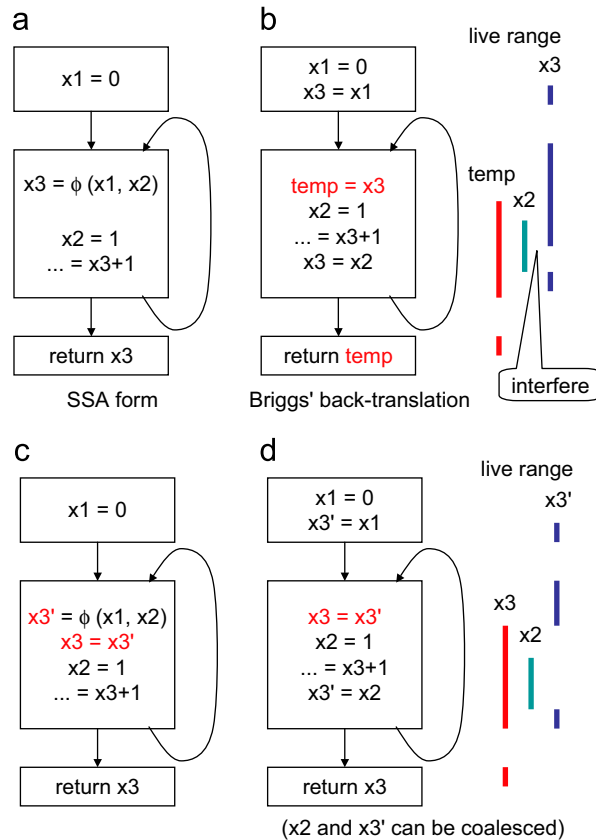
We first make a preliminary comparison of the two algorithms by hand-simulating their behaviors for typical problems. The numbers of copy statements after back-translation by the two algorithms are shown in Table 1.

We can see from the table that in the swap problem, Briggs' algorithm together with coalescing leaves more copy statements in the loop than Sreedhar's algorithm. Considering this result, we will investigate their behavior and will make experiments using benchmarks.

Table 1

Number of copy statements in SSA form and after back-translation (figures in parentheses show the numbers of copy statements in loops)

	SSA form	Briggs	Briggs + coalescing	Sreedhar
Lost copy	0	3	1 (1)	1 (1)
Simple ordering	0	5	2 (2)	2 (2)
Swap	0	7	5 (5)	3 (3)

Fig. 9. Example with unnecessarily long live range. (a) SSA form, (b) Briggs' back-translation, (c) rewrite  $\phi$  and (d) after back-translation.

## 5.2. Drawback in Briggs' algorithm and proposal for improvement

### 5.2.1. Unnecessarily long live ranges

In Briggs' algorithm, copy statements such as " $\text{temp} = \text{target}$ " are sometimes inserted to save the value of the target of a  $\phi$ -function. This often causes related variables to have unnecessarily long live ranges.

In Fig. 9(b), for example, although the value of  $x3$  is stored in  $\text{temp}$ , the live range of  $x3$  continues up to " $\dots = x3 + 1$ ", because  $x3$  is used there. This unnecessarily long live range causes  $x3$  to interfere with  $x2$ . In this case, if we make an improvement that first rewrites the target  $x3$  of the  $\phi$ -function of Fig. 9(a) as in Fig. 9(c) and then perform Briggs' SSA back-translation, we obtain Fig. 9(d). (Rewriting the target of a  $\phi$ -function uses the same method as in Sreedhar's algorithm.) In Fig. 9(d), we can coalesce  $x2$  and  $x3'$  and delete " $x3' = x2$ ".

This improvement may be very effective because saving the target always occurs in loops.

### 5.3. Weakness in Sreedhar's algorithm

Sreedhar's algorithm deletes  $\phi$ -functions by uniting all parameters of a  $\phi$ -function into the same variable. Therefore, by regarding the  $\phi$ -function as a group of copy statements, we can analyze it in the same way as the coalescing in the register allocation phase. That is, the live ranges of the united variables become longer, which may be a disadvantage when the number of allocatable registers is small.

We will investigate this problem in Sections 6 and 7 through experiments.

## 6. Preliminary experiments

### 6.1. Purpose and criteria of evaluation

The purpose of this set of experiments was to investigate the general tendencies of the three back-translation algorithms, Briggs', Sreedhar's and our proposed improvement. To achieve a generally applicable result, we experimented by changing the number of registers and the combination of optimizations.

We used the C-to-SPARC compiler of the COINS compiler infrastructure [16]. Comparisons were made on the framework of the SSA module of COINS [17].

The benchmarks were C programs from SPEC CINT2000.

The processing by this compiler is as follows:

1. The source program is first converted into a pruned SSA form [7] intermediate representation.
2. Several optimizations are applied.
3. The SSA form is back-translated into normal form using one of Briggs', Sreedhar's or our proposed algorithm.
4. Register allocation with iterated register coalescing [18] is applied.
5. The object code is generated.

Instruction scheduling in COINS is at the test stage and does not run for some benchmarks used in this experiment, so we do not use instruction scheduling.

The experiments were run on a Sun Blade 1000. The main specifications are shown in Table 2.

Our experiments used 20 and eight registers. The case of 20 registers is the model for architectures such as SPARC and MIPS, while the use of eight registers investigates the tendency for CPUs in embedded systems to have few registers.

The combinations of optimizations used are shown in Table 3. "opt1" in the graphs of this section corresponds to "opt1" in Table 3, and so on.

Table 2  
Main specifications of Sun Blade 1000

Architecture	Superscalar SPARC V9
Processor	UltraSPARC-III 750 MHz $\times$ 2
L1 cache (KB)	64 (data) 32 (instruction)
L2 cache	8 MB external cache
Memory (GB)	1
OS	SunOS 5.8

Table 3  
Combination of optimizations

opt1	Copy propagation
opt2	Copy propagation, dead code elimination, common subexpression elimination
opt3	Copy propagation, loop invariant code motion

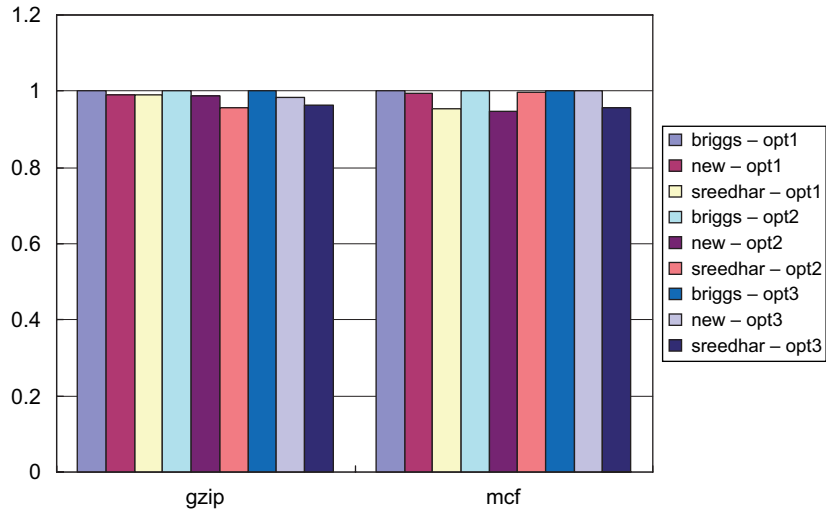


Fig. 10. Ratios of execution times with eight registers.

For benchmarks, we used `gzip` and `mcf` from SPEC CINT2000.

## 6.2. Comparison of execution times

We measured several aspects, such as the number of copy statements that cannot be coalesced, the number (dynamic count) of executed load and store instructions at runtime and the execution time of the object code.

However, because this is a preliminary experiment, we show only the comparison of execution times.

### 6.2.1. With eight registers

Fig. 10 shows the relative ratios of the execution times of the object code when the number of registers is limited to eight. (Small values are better; the reference value is the execution time of Briggs' method, which is normalized to one.) In the figure, "new" means our proposed algorithm for improving Briggs' method (see Section 5.2).

We can see that the execution time with Sreedhar's method is generally shorter than that with Briggs' method, by 3–4% in `gzip` except with optimization 1, and also shorter by 4–5% in `mcf` except with optimization 2.

Our proposed improvement also raised the performance a little compared with Briggs' method, by 1% in `gzip` and by a few percent in `mcf` except with optimization 3.

### 6.2.2. With 20 registers

Next, Fig. 11 shows the relative ratios of execution times when the number of registers is 20. We can see that Sreedhar's method generally produces faster code than Briggs' method, by 1% in `gzip` and by 2–7% in `mcf`.

Our proposed improvement also raised the performance compared with Briggs' method, by 1% in `gzip` and by 1–3% in `mcf`.

From these experiments we realize that Sreedhar's method is superior to Briggs' method in terms of the execution time of the object code. Our proposal for improving Briggs' method was effective to a certain degree, but it could not overcome the predominance of Sreedhar's method. Detailed analysis can be found in [19]. This thesis also shows that other measurements such as the number of copy statements that cannot be coalesced and the number (dynamic count) of executed load and store instructions at runtime give the same tendency, that is, our method is slightly better than Briggs' method, but it could not overcome the predominance of Sreedhar's method. Therefore, in the following section we focus on Briggs' method and Sreedhar's method and make more detailed evaluations using further benchmarks.

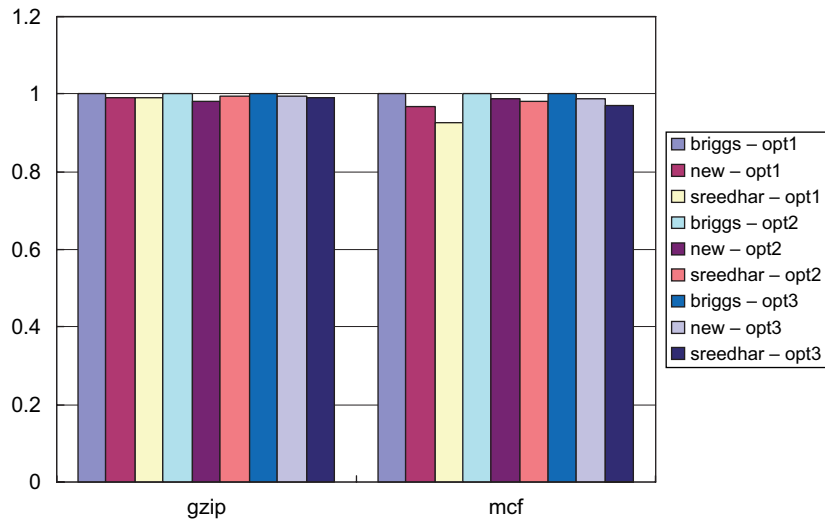


Fig. 11. Ratios of execution times with 20 registers.

Table 4  
Combinations of optimizations

Optimization 0	No optimization
Optimization 1	Copy propagation, conditional constant propagation [20], dead code elimination, empty block elimination
Optimization 2	Remove critical edge, loop invariant code motion, conditional constant propagation [20], common subexpression elimination, copy propagation, conditional constant propagation [20], dead code elimination, empty block elimination

## 7. Detailed experiments

### 7.1. Purpose and criteria of evaluation

The purpose of this set of experiments was to focus on Briggs’ method and Sreedhar’s method, and make a more detailed evaluation than in the previous section, using more benchmarks.

We used eight and 20 registers, as in the previous section. The combinations of optimizations are shown in Table 4. (The optimizations applied are different from those in the previous section.) In the graphs in this section, we denote “no optimization” as “-opt0”, “optimization 1” as “-opt1” and “optimization 2” as “-opt2”.

From our previous discussion, we can anticipate that in Briggs’ method, many copy statements are inserted that cannot be coalesced during register allocation (see Section 5.2). Copy statements that cannot be coalesced may possibly waste registers unnecessarily.

On the other hand, in Sreedhar’s method, we can anticipate similar problems with the aggressive coalescing in register allocation (see Section 5.3).

Therefore, we first made a *static* evaluation by measuring the number of move instructions (the number of copy statements that cannot be coalesced) in the object code, and the number of variables that are spilled in the register

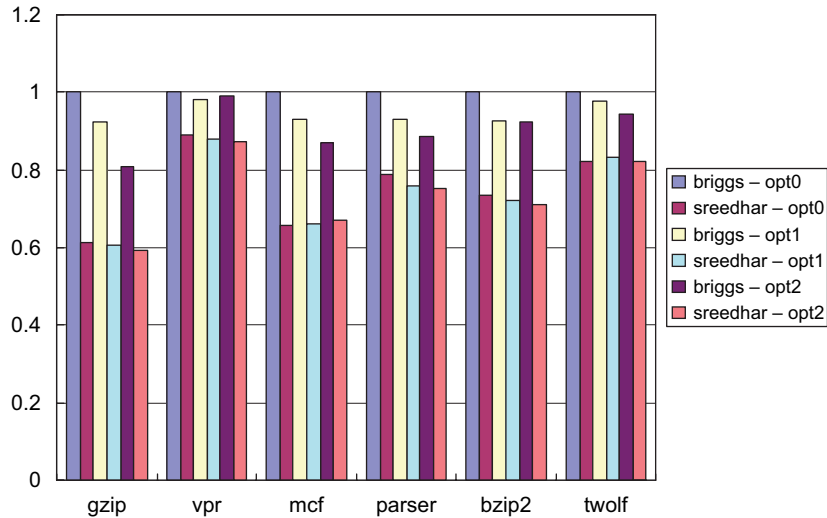


Fig. 12. Ratios of the static numbers of moves with eight registers.

allocation phase. This gives insight into how copy statements that cannot be coalesced affect the object code and register allocation. (The *static* number in compilers means the number at compilation time. It contrasts with the *dynamic* number, which represents the number at execution time.) Next, we made a *dynamic* evaluation by measuring the number of executed move instructions, and the number of executed load and store instructions. This is to check the actual influence of static measurements on runtime measurements.

We finally measured the execution time as the final criterion.

For benchmarks, we used `gzip`, `vpr`, `mcf`, `parser`, `bzip2` and `twolf` from SPEC CINT2000. The environment of the experiments is the same as for the first experiments.

## 7.2. Comparisons of the static numbers of move instructions in the object code

To examine the differences in the back-translation algorithms, we measured the static number of move instructions in the object code. This measurement helps us understand how the differences in the number of copy statements that cannot be coalesced and the process of uniting parameters in a  $\phi$ -function (making them a single variable in Sreedhar's method) affect the object code.

Note that we counted the static number of move instructions in this section. We therefore cannot judge the relative superiority of algorithms at this stage.

### 7.2.1. With eight registers

Fig. 12 shows the relative ratios of the (static) number of move instructions in the object code when the number of registers is limited to eight. (Small values are better; the reference value is that for no optimization in Briggs' method, which is normalized to one.)

We can see that the number of move instructions in Sreedhar's method is 60–90% of that of Briggs' method. From this, we confirm that in Briggs' method many copy statements that cannot be coalesced are inserted. We can anticipate that the dynamic number of move instructions at execution time will be larger with Briggs' method. This will be the subject of a later experiment.

Comparing the combination of optimizations, we see that in Briggs' method the number of move instructions becomes smaller as we perform more optimizations. This is in contrast to Sreedhar's method where the number does not generally change.

### 7.2.2. With 20 registers

The result of the same experiment for 20 registers is shown in Fig. 13.

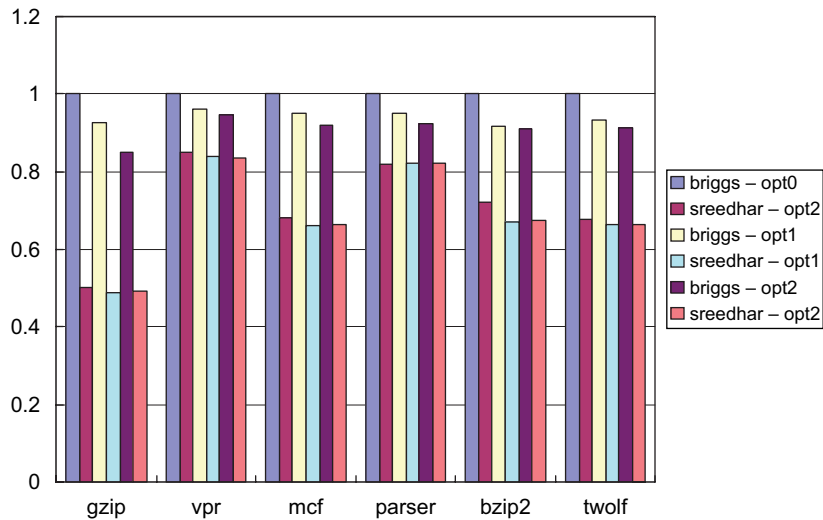


Fig. 13. Ratios of the static numbers of moves with 20 registers.

We can see that the number of move instructions in Sreedhar's method is about 50–80% of that of Briggs' method. The difference is much greater than with eight registers.

When more optimizations are applied, the number of move instructions decreases in Briggs' method, which is similar to the case with eight registers. This is in contrast to Sreedhar's method where the number does not generally change.

### 7.3. Comparison of the number of spills

We counted the (static) number of variables that are spilled at the register allocation phase, to investigate how the differences in SSA back-translation algorithms affect register allocations, through the differences in the number of copy statements that cannot be coalesced, and the uniting process of the parameters in  $\phi$ -functions. Note, however, that we cannot judge the superiority of algorithms at this stage, because this number is static and does not represent the dynamic cost of spills, which is similar to the comparison of the number of move instructions.

#### 7.3.1. With eight registers

Fig. 14 shows the number of variables that are spilled in the register allocation phase when the number of registers is limited to eight.

We can see that fewer variables are spilled with Sreedhar's method than with Briggs' method. We assume this is because registers are uselessly consumed in Briggs' method because of copy statements that cannot be coalesced. From this, we can anticipate that the dynamic number of load and store instructions at execution time will be bigger in Briggs' method. This will be the subject of a later experiment.

#### 7.3.2. With 20 registers

Fig. 15 shows the number of variables that are spilled at the register allocation phase when the number of registers is 20.

We can see that fewer variables are spilled with Sreedhar's method than with Briggs' method, which is similar to the case of eight registers. As well, there are fewer spills than with eight registers. From this, we can anticipate that the influence of spills on the dynamic number of load and store instructions at execution time with 20 registers will be less than with eight registers. We will also confirm this in a later experiment.

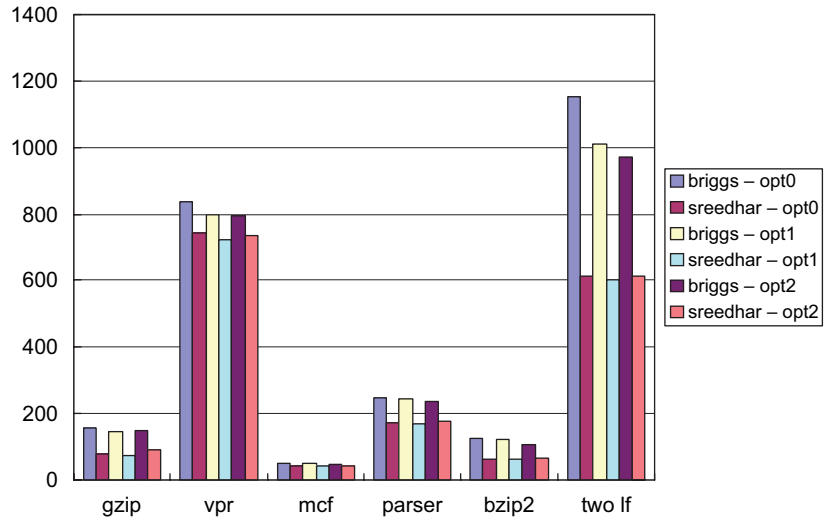


Fig. 14. Static number of spills with eight registers.

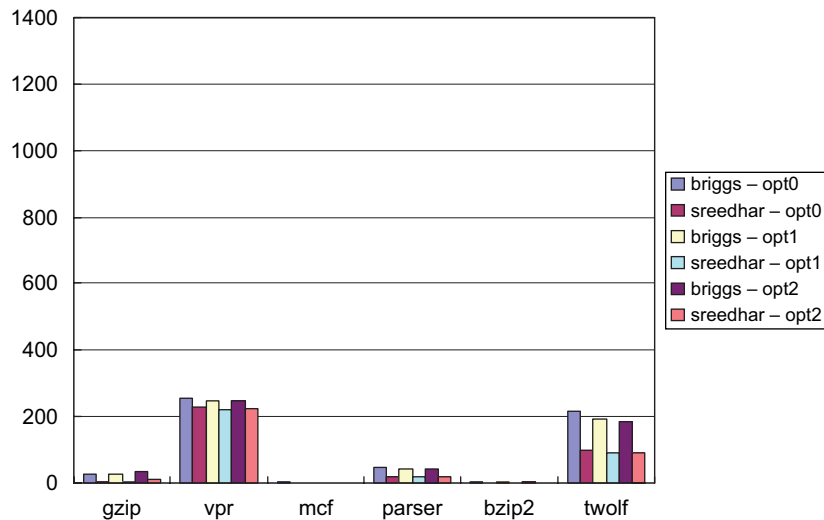


Fig. 15. Static number of spills with 20 registers.

#### 7.4. Comparison of the numbers of executed move instructions

In the static measurements, the number of move instructions in Sreedhar's method was smaller than that in Briggs' method. Here, we measure the dynamic count of move instructions of the object code at execution time.

##### 7.4.1. With eight registers

Fig. 16 shows the relative ratios of the dynamic counts of executed move instructions when the number of registers is limited to eight. (Small values are better; the reference value is the case of no optimization in Briggs' method, which is normalized to one.)

We can see that the dynamic count of executed move instructions in Sreedhar's method is 10% of that of Briggs' method in gzip and about 50–80% of that of Briggs' method in others.

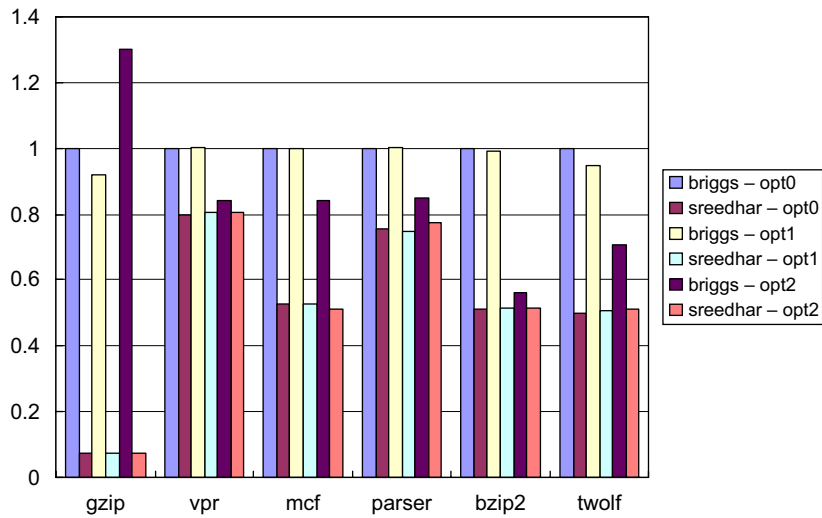


Fig. 16. Ratios of the numbers of executed moves with eight registers.

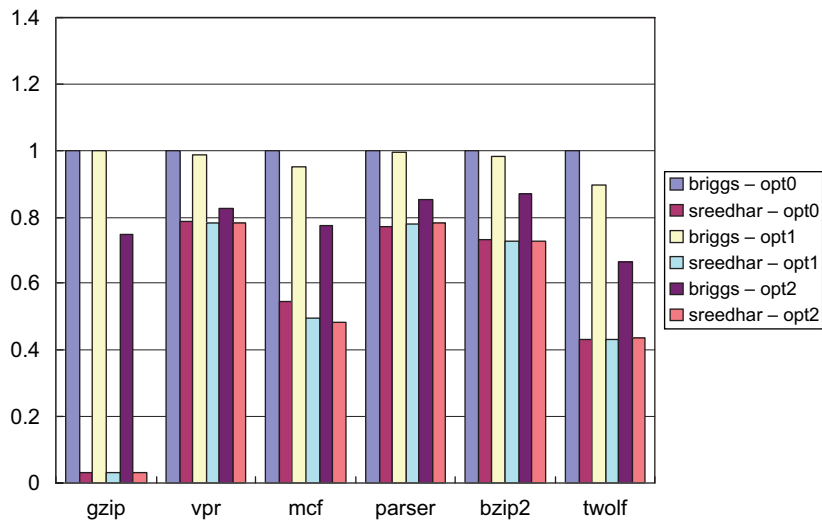


Fig. 17. Ratios of the numbers of executed moves with 20 registers.

Changing the combination of optimizations shows the same tendencies as in the static measurements. The number of executed move instructions does not generally change by optimizations in Sreedhar’s method.

#### 7.4.2. With 20 registers

Fig. 17 shows the relative ratios of the dynamic counts of executed move instructions when the number of registers is 20.

We can see that the dynamic count of executed move instructions in Sreedhar’s method is 3% in `gzip` and about 40–80% in others, compared with that in Briggs’ method. This shows greater difference than with eight registers, which is similar to the static measurement.

Comparison by changing the combination of optimizations shows that the number of executed move instructions in Sreedhar’s method does not generally change.

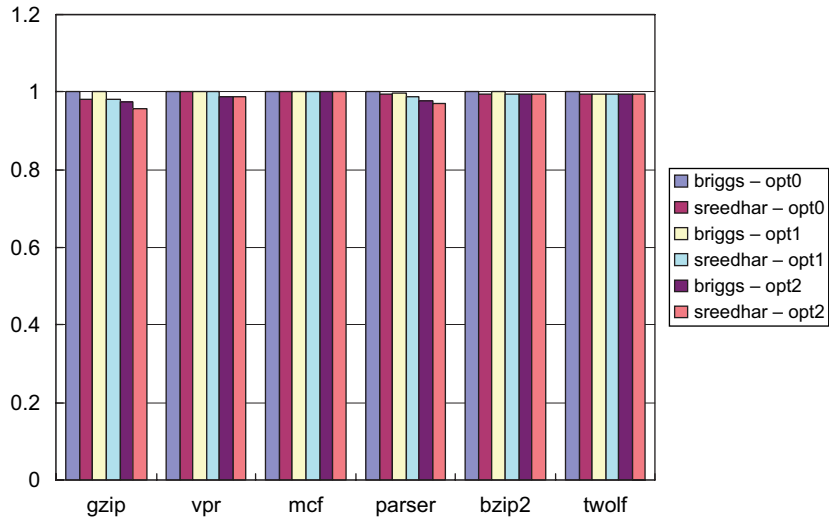


Fig. 18. Ratios of the numbers of executed load and store instructions with eight registers.

### 7.5. Comparison of the numbers of executed load and store instructions

In static measurements, Sreedhar's method has fewer spills. However, in Sreedhar's method, the parameters of the  $\phi$ -functions are united. This may lengthen the live range of variables involved in  $\phi$ -functions, and may result in increasing the cost of spills per variable.

Therefore, we measured the dynamic counts of executed load and store instructions of the object code, as an approximation of the cost of spills at execution time.

#### 7.5.1. With eight registers

Fig. 18 shows the relative ratios of the dynamic counts of executed load and store instructions of the object code when the number of registers is limited to eight. (Small values are better; the reference value is the case of no optimization in Briggs' method, which is normalized to one.)

We can see that Sreedhar's method is favorable in four out of six benchmarks while Briggs' method is favorable in two of the benchmarks. In particular, Sreedhar's method is quite advantageous in `gzip`.

Moreover, although the number of spilled variables was absolutely smaller with Sreedhar's method than with Briggs' method, the difference in the numbers of executed load and store instructions in both methods is reduced or partly reversed. Therefore, we can assume that the cost of spills per variable is higher with Sreedhar's method. Even if the combination of optimizations is changed, the difference in the numbers of executed load and store instructions in both methods is preserved.

#### 7.5.2. With 20 registers

Fig. 19 shows the relative ratios of the dynamic counts of executed load and store instructions with 20 registers. Little difference is found, as opposed to the case of eight registers. This is because the difference in the numbers of spilled variables in Fig. 15 is smaller than in Fig. 14.

Thus, because there is almost no difference in the numbers of executed load and store instructions in both methods, we expect that the execution time with 20 registers depends mainly on the difference of the numbers of executed move instructions.

### 7.6. Comparison of execution times

Lastly, we present a comparison of execution times of the object codes.

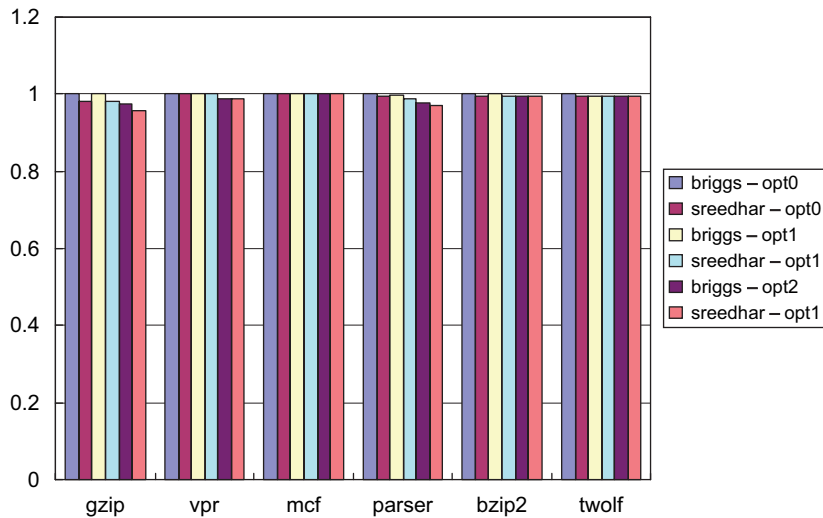


Fig. 19. Ratios of the numbers of executed load and store instructions with 20 registers.

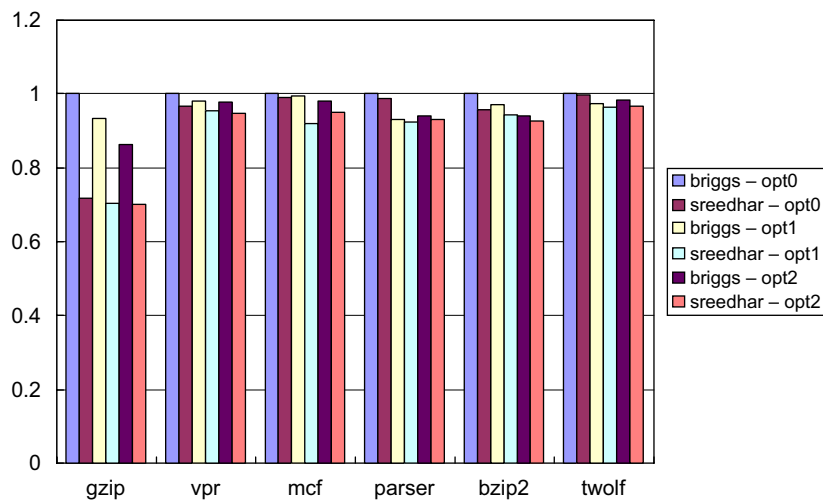


Fig. 20. Ratios of execution times with eight registers.

### 7.6.1. With eight registers

Fig. 20 shows the relative ratios of the execution times with eight registers. (Small values are better; the reference value is the case of no optimization in Briggs' method, which is normalized to one.)

The execution speed of the object code generated with Sreedhar's method is generally faster than that with Briggs' method, by about 28% in `gzip`, and a little faster in all the others. We discuss scattering at execution time and the relative ratios in some detail later.

### 7.6.2. With 20 registers

Next, Fig. 21 shows the relative ratios of the execution times with 20 registers. The execution speed of the object code from Sreedhar's method is faster in most benchmarks by as much as 8% compared with that from Briggs' method. The exceptions are with all combinations of optimizations in `mcf` and the no-optimization case in `parser`, where the object code from Sreedhar's method is slower by about 3%.

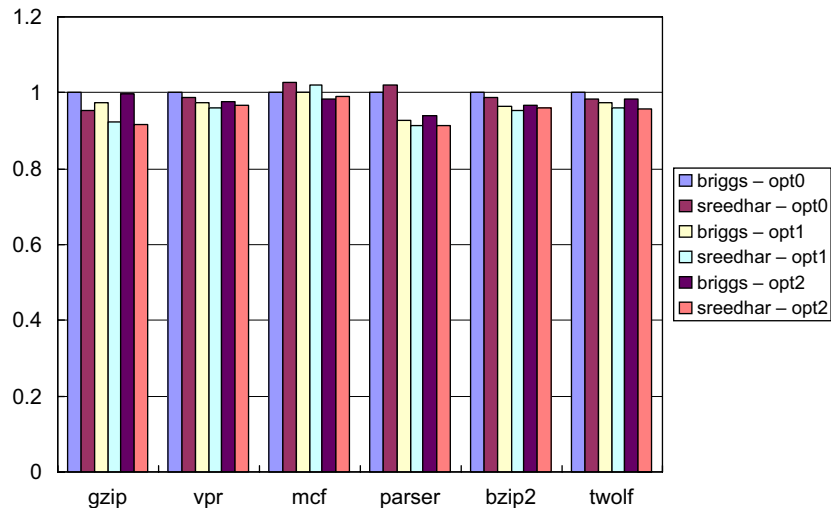


Fig. 21. Ratios of execution times with 20 registers.

In the comparison of execution times, the results mostly coincide with the previous analysis. However, the results do differ partly in showing some fluctuation or scattering.

We think that these fluctuations of the results are mainly because of the characteristics of the target architecture such as superscalar and cache lines, rather than the effect of SSA back-translation algorithms.

One possible reason relates to instruction fetch cycles. For example, if there is a branch instruction in a loop, the processor tries to fetch a sequence of next instructions using branch prediction while processing this branch instruction. If instructions to be fetched reside in one cache line, they can be fetched together at once. If they straddle the boundaries of cache lines, however, fetching takes extra time.

In fact, when we re-aligned instructions in a loop that takes most of the execution time according to the cache lines, the execution time sometimes decreases by 4–5% [19]. This kind of internal processing mechanism has particular influence in superscalar machines, such as the one used in this experiment.

Therefore, for the execution time of the object code, Sreedhar's method is favorable on average, although in some cases the effect of the difference of back-translation algorithms is hidden by other factors.

In summary, with eight registers, Sreedhar's method achieves shorter execution times of the object code than Briggs' method. This is considered to be because of the smaller dynamic cost of spills and the smaller number of executed move instructions.

On the other hand, with 20 registers, Sreedhar's method achieves shorter execution time of the object code in most cases, although there are a few exceptions. This is considered to be because of the smaller number of executed move instructions required.

Moreover, the differences in SSA back-translation algorithms are not generally influenced by changing the combination of optimizations.

## 8. Related work

Our work aims at empirically comparing SSA back-translation algorithms. As far as we know, apart from Rastello et al. [14], there have been no similar studies performed previously. Therefore, in this section we present other SSA back-translation algorithms and discuss Rastello et al.'s work.

### 8.1. Morgan's SSA back-translation

First, we briefly explain Morgan's back-translation algorithm [13]. This algorithm replaces  $\phi$ -functions by copy statements, similar to Briggs' method.

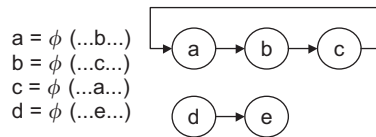


Fig. 22. Morgan's method.

In Morgan's method, a graph like Fig. 22 is used. In this graph, nodes represent variables, and directed edges are drawn from the targets (left-hand side variables) to the parameters of the  $\phi$ -functions.

In this graph, variables (nodes) that have no predecessor nodes do not appear in the parameters of  $\phi$ -functions. This means that such a variable can be overwritten by back-translation, and a copy statement "that variable = variable of its successor node" can be safely inserted.

In addition, if this graph has a cycle, it will lead to dependency among copy statements to be inserted, which must be resolved. The cyclic dependency is cut by using a temporary variable.

For example, in Fig. 22, because  $d$  has no predecessor node, " $d = e$ " can be inserted. Next, because  $a \rightarrow b \rightarrow c \rightarrow a$  has a cycle, we first insert " $temp = a$ " to save the value of  $a$ , and then insert copy statements along the directed edges. The resulting copy statements are as follows:

$$\begin{aligned} d &= e, \\ temp &= a, \\ a &= b, \\ b &= c, \\ c &= temp. \end{aligned}$$

Briggs' method can also deal with such cases, and in certain forms of cycles, Briggs' method inserts fewer copy statements. Therefore, Morgan's method is actually a subset of Briggs' method.

### 8.2. SSA back-translation by Budimlic et al.

Unlike the algorithms that are the subjects of our experiments, Budimlic et al. [21] were mainly concerned with reducing compilation time while preserving precision.

They presented a fast algorithm for treating interference among variables in a program without constructing an interference graph. They then described how to use this information to minimize copy insertions in SSA back-translation, yielding a fast copy coalescing. They proved some properties of the SSA form that enable computation of interference information for variables that are considered for folding. The live ranges were approximated using these properties.

Performing copy folding during the SSA back-translation eliminates the need for a separate coalescing phase. This may make graph-coloring register allocation more practical in just-in-time compilers.

In their method, the "lost copy problem" is avoided by splitting critical edges, as opposed to Briggs' or Sreedhar's algorithms, which do not require edge splitting.

They presented experimental results demonstrating that their algorithm is faster than and almost as precise as Briggs' back-translation followed by Briggs-style interference-graph-based coalescing, although the number of copy statements after coalescing varies significantly.

In summary, their work is effective and gives an efficient algorithm similar to Briggs' back-translation and coalescing. However, it does not improve the translated result.

### 8.3. SSA back-translation by Rastello et al.

Rastello et al. developed a back-translation algorithm for machine instruction-level language, under the renaming constraints of machine-dedicated registers. These constraints mean that registers specified by special instructions or the application binary interface (ABI), such as the stack pointer or function parameter registers, must be in specific registers. The machine-level language before back-translation allows symbolic register names to be used freely.

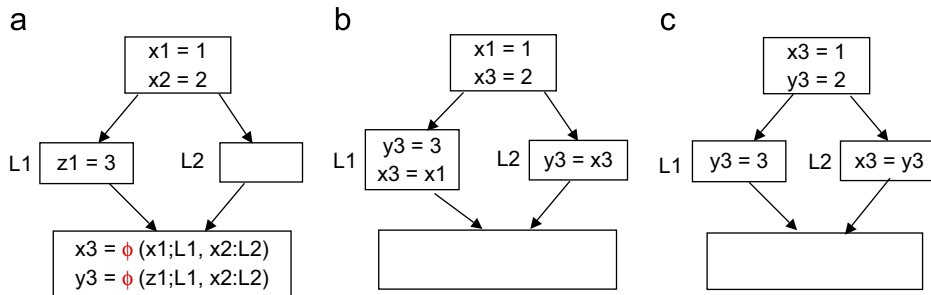


Fig. 23. Example of Rastello's and Sreedhar's back-translation. (a) SSA form, (b) back-translation by Sreedhar et al. and (c) back-translation by Rastello et al.

Rastello et al.'s method is an improvement on the algorithm of Leung and George that handles renaming constraints [22], which is in turn based on Briggs' back-translation algorithm.

They performed the so-called *pinning* (precoloring of variables to physical registers or variables) of  $\phi$  arguments and their corresponding definition to a common resource, and reduced the number of  $\phi$ -related copy statements during the SSA back-translation. This has the effect of coalescing.

They defined the  $\phi$  *coalescing problem* to be the problem of finding a variable pinning that maximizes the total gain of  $\phi$  coalescing, taking into account all  $\phi$ -instructions in the program. The  $\phi$  coalescing problem was shown to be NP-complete, so they gave an approximate algorithm using a sequence of local optimizations that is heuristic and greedy. In a sense, this resembles Sreedhar's algorithm.

After SSA back-translation, register allocation of nondedicated variables was performed in a "repeated coalescing register allocator", which is an improvement on the iterated register coalescing of George and Appel [18].

Rastello et al. asserted that combining coalescing and ABI constraint renaming during the SSA back-translation gives a better result than that would occur if each handling process was kept separate.

For comparison with Sreedhar's algorithm, they claimed the following:

- Sreedhar's algorithm treats each  $\phi$ -instruction in sequence. Rastello et al.'s algorithm considers all the  $\phi$ -instructions of a given block together. This can lead to a better solution in some cases as in the example of Fig. 23.
- Because their SSA back-translation is made at machine level, it can give a better result by taking into account ABI constraints at the same time, compared with Sreedhar's algorithm, where ABI constraints must be treated afterwards.

They implemented their algorithm for a DSP processor, and performed several experiments using an aggressive register coalescing. To compare with Sreedhar's method, they made a module that simulated it within their framework, but this may be incorrect in some situations, as they note.

For comparison of the algorithms without ABI constraints, the number of (static) move instructions by their method for SPEC CINT (probably the sum for all benchmarks in it, but the data of each benchmark are not given) was 6803, compared with 6744 in Sreedhar's method, which means their method is slightly inferior. With ABI constraints, the number of (static) move instructions by their method in SPEC CINT is 23,930, compared with 24,343 in Sreedhar's method, which means theirs is a little better.

From these experiments, it seems difficult to judge relative superiority, because there is no comparison of the dynamic numbers of move instructions nor a comparison of execution times of object code. We assume that their method yields a slightly inferior quality than Sreedhar's method does if SSA back-translation is to be performed without ABI constraints (as far as SPEC CINT is concerned). We also assume that their method is advantageous if SSA back-translation is to be performed with ABI constraints at machine language level, although machine-language-level intermediate representation is not very common.

In summary, their contribution is that they showed that considering SSA back-translation and renaming constraints of dedicated registers together results in an improved coalescing of variables, reducing the number of move instructions before register allocation.

## 9. Discussion

### 9.1. Instruction scheduling

In our experiments, no instruction scheduling was performed, as described in Section 6.1. However, as most modern processors provide instruction-level parallel processing, instruction scheduling's influence on performance is not negligible. In addition, it is known that instruction scheduling and register allocation interfere with each other. That is, if we decrease the number of actual registers in the register allocation, the parallelism between instructions is hampered. On the other hand, if we enhance parallelism by instruction scheduling, the register pressure will increase.

Because register allocation and instruction scheduling depend on each other, many studies have been performed concerning the order of their application and their cooperative processing [23].

The consequence for our experiments is that if we increase the reusability of registers by coalescing, the dependencies between instructions will increase, and this may decrease the instruction-level parallelism. In processors performing out-of-order execution, this may further hinder the exchange of instructions. Therefore, evaluation using instruction scheduling will be necessary in the future.

### 9.2. Coalescing in register allocation

Chaitin's original coalescing register allocation [15] aggressively eliminates all copy statements by coalescing the source and the target nodes of a copy if they do not interfere in the interference graph. However, it is known that this may be harmful to the colorability of the graph and may cause spills. Therefore, many coalescing algorithms for register allocation have been studied.

Park et al. proposed *optimistic register coalescing* [24], which optimistically performs aggressive coalescing, thereby exploiting the positive impact of coalescing aggressively, but when a coalesced node is to be spilled, it is split back into separate nodes. Therefore, this approach can reduce the overall spill rate while performing aggressive coalescing.

They gave the results of several experiments. For example, iterated coalescing by George et al. [18] and optimistic coalescing by Park et al. removed a geometric mean of 75.5% and 98.9%, respectively, of the copy statements removed by the original aggressive coalescing by Chaitin. This means that the static numbers of move instructions in Briggs' method in Figs. 12 and 13 may be reduced by adopting optimistic coalescing. Because coalescing and spills are interrelated, we cannot anticipate optimistic coalescing's effect on the number of spills, the number of load and store instructions or the execution time.

Park et al. also gave figures for spills and execution cycles on a VLIW machine. In the comparison of dynamic spill instructions and execution cycles between iterated coalescing and optimistic coalescing, there was some variance, and iterated coalescing gave better results than optimistic coalescing in two out of the seven benchmarks, whereas the latter was better in the other five.

We note in Section 4.2 that in Sreedhar's algorithm, any copy statements that it inserts cannot be eliminated by the standard interference-graph-based coalescing algorithm, but the long live ranges made by uniting sources and destination of  $\phi$ -functions may be a problem. Using live range splitting as in Park et al.'s algorithm may improve register pressure. Experiments using optimistic register coalescing by Park et al. will be the subject of future work.

From a different point of view, simpler register allocation is used in just-in-time compilers. Experimenting with the trade-off between aggressive register allocations and nonaggressive register allocations such as the linear scan register allocation [25] used in just-in-time compilers will also be an interesting future study (see also Section 8.2).

### 9.3. Back-translation with dedicated registers

We surveyed the SSA back-translation algorithm by Rastello et al. under the constraints of dedicated registers in Section 8.3. We indicated there that Sreedhar's algorithm is still better than theirs without the constraints of dedicated registers (as far as SPEC CINT is concerned). However, in embedded processors, such as DSP processors, their algorithm appears to be more favorable. It would be an interesting exercise to compare SSA back-translation algorithms under such constraints.

## 10. Conclusions

Two major algorithms exist for the back-translation from SSA form to normal form. However, there have been no previous studies that empirically compare them and investigate their characteristics in depth. As a result, many compilers using the SSA form simply adopted the method by Briggs et al. without much consideration, because it was the first solution of the critical problems of the early naive algorithm.

In this paper, we clarified the merits and demerits of different SSA back-translation algorithms, and validated them through experiments. We showed that the selection of the back-translation algorithm affects the runtime efficiency of the object code in no small way, which has not previously received attention. The effect is comparable to applying a middle-level global optimization. A major contribution of our work was to give criteria for selecting the SSA back-translation algorithm in future compilers.

The main results in our experimental environment are as follows:

- In Briggs' method, a large number of copy statements that cannot be coalesced are inserted. They affect the performance of the object code more than they increase the register pressure, which is a concern in Sreedhar's method.
- When there are relatively few allocatable registers, Sreedhar's method, which performs uniting on the  $\phi$ -functions, is superior because of the reduced dynamic cost of spills and the reduced number of executed copy statements. Its execution time is better than Briggs' method by a few percent in general, and by 28% at maximum.
- When there are relatively many allocatable registers, Sreedhar's method is favorable in most cases, because the number of copy statements executed is low. Its execution time is better than Briggs' method by a few percent in most cases.

Further experiments by changing the combination of compiler phases, such as optimistic register coalescing, back-translation with dedicated registers and instruction scheduling, and their analysis are left for future research.

## Acknowledgments

This research was partially supported by the Japanese Ministry of Education, Culture, Sports, Science and Technology under the Grant "Special Coordination Fund for Promoting Science and Technology" and by the Japan Society for the Promotion of Science under a Grant-in-Aid for Scientific Research and by the Kayamori Foundation for Informational Science Advancement.

## References

- [1] Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 1991;13(4):451–90.
- [2] Fitzgerald R, Knoblock TB, Ruf E, Steensgaard B, Tarditi D. Marmot: an optimizing compiler for Java. *Software—Practice and Experience* 2000;30(3):199–232.
- [3] GCC. GCC homepage. (<http://gcc.gnu.org/>).
- [4] IBM. Jikes research virtual machine. (<http://jikesrvm.sourceforge.net/>).
- [5] MachSUIF homepage. (<http://www.eecs.harvard.edu/machsuiif/>).
- [6] Scale Compiler Group. University of Massachusetts. (<http://www-ali.cs.umass.edu/Scale/>).
- [7] Briggs P, Cooper KD, Harvey TJ, Simpson LT. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience* 1998;28(8):859–81.
- [8] Briggs P, Harvey TJ, Simpson LT. Static single assignment construction, version 1.0. (<ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps>), January 1996.
- [9] Sreedhar VC, Ju RD-C, Gillies DM, Santhanam V. Translating out of static single assignment form. In: *Proceedings of the sixth international symposium on 37 static analysis. Lecture notes in computer science*, vol. 1694. Berlin: Springer; 1999. p. 194–210.
- [10] Ishizaki K, Takeuchi M, et al. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. In: *OOPSLA '03: Proceedings of the 18th ACM 36 SIGPLAN conference on object-oriented programming, systems, languages, and applications*, 2003. p. 187–204.
- [11] Appel AW. *Modern compiler implementation in Java*. 2nd ed., Cambridge: Cambridge University Press; 2002.
- [12] Sreedhar VC, Gao GR. A linear time algorithm for placing  $\phi$ -nodes. In: *Proceedings of the 22nd ACM SIGPLAN–SIGACT symposium on principles of programming languages*. 1995. p. 62–73.
- [13] Morgan R. *Building an optimizing compiler*. Belford, MA: Digital Press; 1998.
- [14] Rastello F, de Ferrière F, Guillon C. Optimizing translation out of SSA using renaming constraints. In: *Proceedings of the international symposium on code generation and optimization (CGO '04)*. 2004. p. 265–78.

- [15] Chaitin GJ. Register allocation & spilling via graph coloring. In: Proceedings of the SIGPLAN '82 symposium on compiler construction. 1982. p. 98–105.
- [16] The coins project. Research of a common infrastructure for parallelizing compilers. (<http://www.coins-project.org/>).
- [17] Sassa M, Nakaya T, Kohama M, Fukuoka T, Takahashi M. Static single assignment form in the COINS compiler infrastructure. In: Proceedings of SSGRR 2003w—international conference on advances in infrastructure for e-business, e-education, e-science, e-medicine, and mobile technologies on the internet, 2003.
- [18] George L, Appel AW. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems* 1996;18(3):300–24.
- [19] Kohama M. Comparison and evaluation of SSA normalization algorithms. Master's thesis, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2004. (<http://www.is.titech.ac.jp/%7Esassa/lab/papers-written/02M37130.pdf>) (in Japanese).
- [20] Wegman MN, Zadeck FK. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* 1991;13(4):181–210.
- [21] Budimic Z, Cooper KD, Harvey TJ, Kennedy K, Oberg TS, Reeves SW. Fast copy coalescing and live-range identification. In: Proceedings of the ACM SIGPLAN 2002 conference on programming language design and implementation. 2002. p. 25–32.
- [22] Leung AL, George L. Static single assignment form for machine code. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation. 1999. p. 204–14.
- [23] Inagaki T, Komatsu H, Nakatani T. Integrated prepass scheduling for a Java just-in-time compiler on the IA-64 architecture. In: Proceedings of international symposium on code generation and optimization: feedback-directed and runtime optimization. 2003. p. 159–68.
- [24] Park J, Moon S-M. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems* 2004;26(4):735–65.
- [25] Poletto M, Sarkar V. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems* 1999;21(5):895–913.

**Masataka Sassa** is a professor in Computing Science at the Tokyo Institute of Technology in Japan. His research interests include the design and implementation of programming languages and programming environments.

**Yo Ito** has been a graduate student in Computing Science at the Tokyo Institute of Technology. His research interests include the implementation of programming languages.

**Masaki Kohama** has been a graduate student in Computing Science at the Tokyo Institute of Technology. His research interests include the implementation of programming languages.